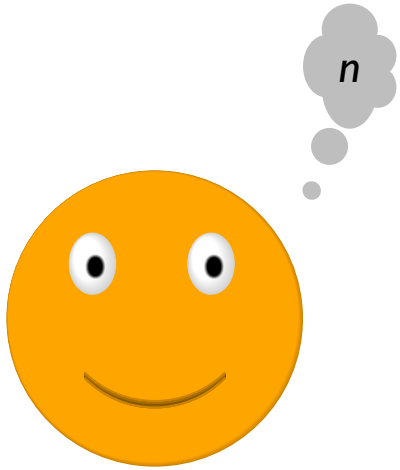
Guess The Number

I'm thinking of a number between 0 and 100

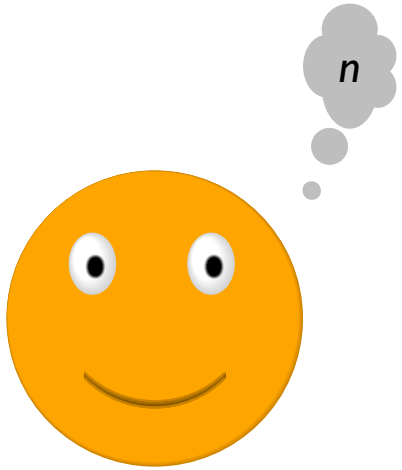


Guess, and I'll tell you whether you're right or wrong

How many guesses do you need?

Guess The Number

I'm thinking of a number between 0 and 100



For each wrong guess you make, I'm willing to answer with *higher* or *lower*

How many guesses do you need?

Binary Search

The optimal guessing strategy for a number between *n* and *m*:

- If *n* equals *m*, guess *n*
  - You win!
- Guess *p* = `(quotient (+ n m) 2)`
  - If *p* is too high, apply optimal strategy for *n* to *p*
  - If *p* is too low, apply optimal strategy for `(add1 p)` to *m*
  - Otherwise, you won!

This kind of strategy is called ***divide and conquer***

Binary Search

```
; A dir is either 'too-big, 'too-small, or 'same

; bsearch num num (num -> dir) -> num
(define (bsearch n m guess)
  (cond
    [(= n m) n]
    [else
     (define p (quotient (+ n m) 2))
     (define d (guess p))
     (cond
       [(eq? d 'too-big)
        (bsearch n p guess)]
       [(eq? d 'too-small)
        (bsearch (add1 p) m guess)]
       [else p])]))
```

Binary Search in C

```
int bsearch(int n, int m)
{
  if (n == m)
    return n;
  else {
    int p = (n + m) / 2;
    int c = check(p);

    if (c > 0)
      bsearch(n, p);
    else if (c < 0)
      bsearch(p + 1, m);
    else
      return p;
  }
}
```

Binary Search in C: Loop Variant

```
int bsearch_loop(int n, int m)
{
  while (n != m) {
    int p = (n + m) / 2;
    int c = check(p);

    if (c > 0)
      m = p;
    else if (c < 0)
      n = p + 1;
    else
      return p;
  }

  return n;
}
```

Binary Search on a Sorted Array

If you have an array of sorted values, then finding a particular value is a game of "Guess the Number" where the number is an array index

See **`phonebook.rkt`**

See **`bsearch.c`**

Sorting an Array

Remember **`sort-list`**?

```
; sort-list : list-of-num -> list-of-num
(define (sort-list l)
  (cond
    [(empty? l) empty]
    [(cons? l) (insert (first l) (sort-list (rest l)))]))

; insert : num list-of-num -> list-of-num
(define (insert n l)
  (cond
    [(empty? l) (list n)]
    [(cons? l)
     (cond
       [(< n (first l)) (cons n l)]
       [else (cons (first l) (insert n (rest l)))])]))
```

- Insertion is bad for arrays
- Run time was $O(n^2)$ anyway

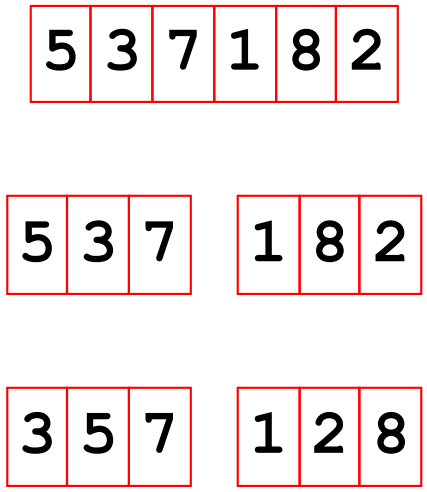
Sorting an Array: Divide and Conquer

Mergesort:

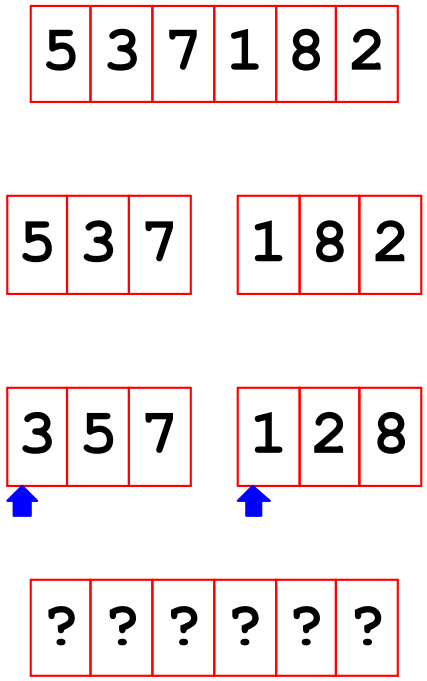
- If the array has one value, you're done
- Otherwise:
  - Split the array into two equal parts
  - Sort the two parts
  - Merge the two sorted parts

$$T(n) = k_1 n + 2T(n/2) + k_2 n$$

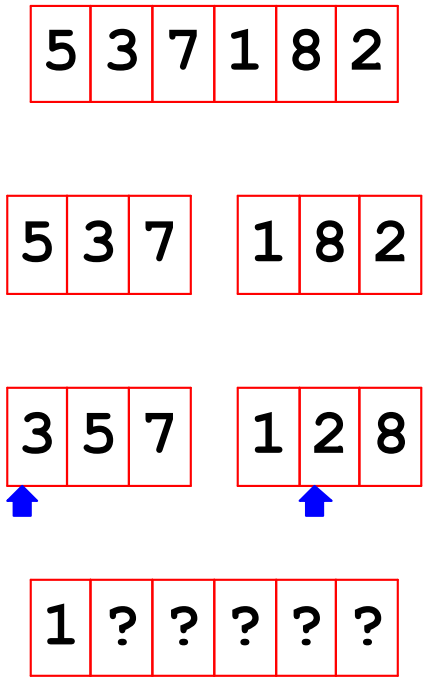
Sorting an Array: Divide and Conquer



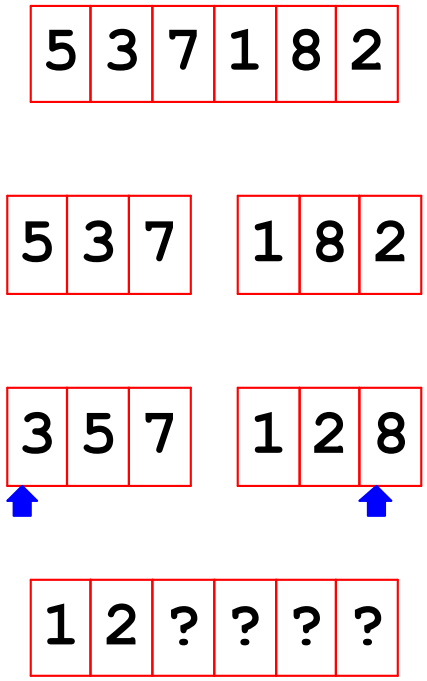
Sorting an Array: Divide and Conquer



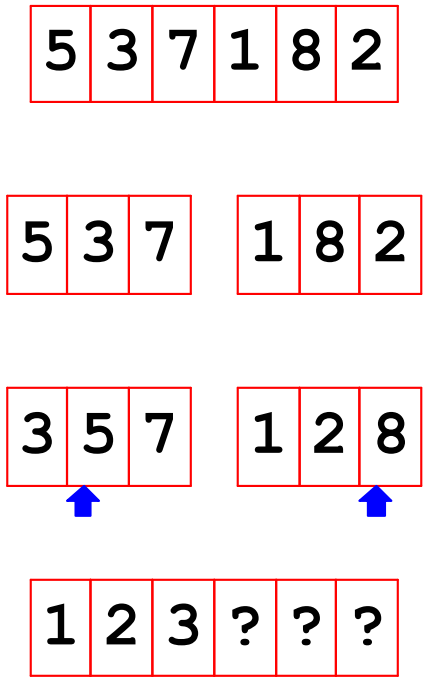
Sorting an Array: Divide and Conquer



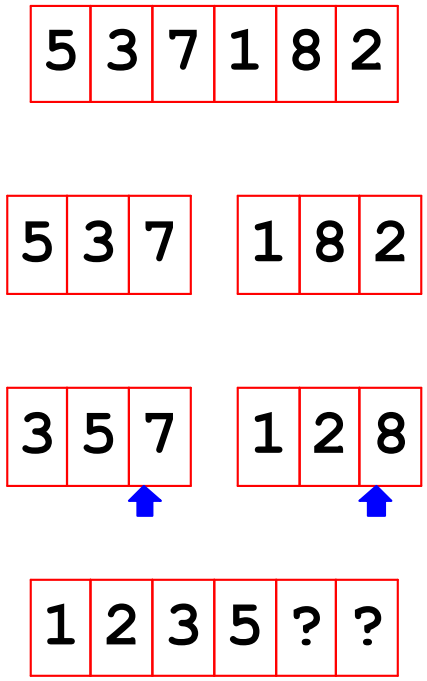
Sorting an Array: Divide and Conquer



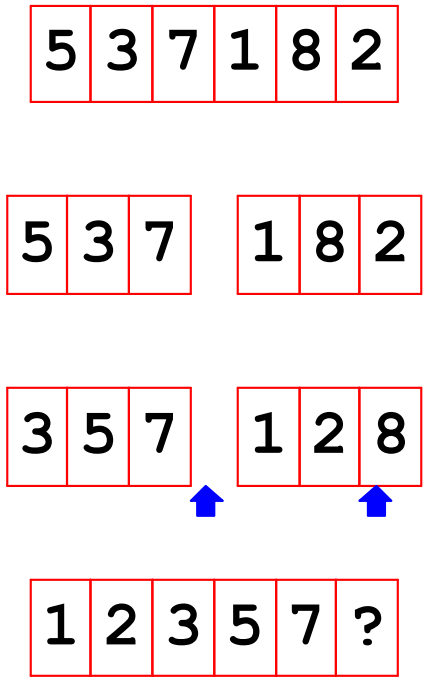
Sorting an Array: Divide and Conquer



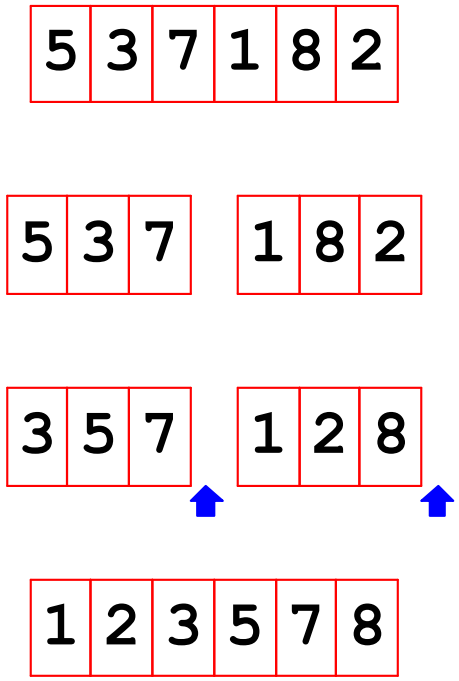
Sorting an Array: Divide and Conquer



Sorting an Array: Divide and Conquer



Sorting an Array: Divide and Conquer



List Mergesort

```
; mergesort: list-of-num -> list-of-num [sorted]
(define (mergesort ls)
  (define len (length ls))
  (cond
    [(<= len 1) ls]
    [else
     (define half (quotient len 2))
     (merge
      (mergesort (take ls half))
      (mergesort (drop ls half)))]))
```

List Merge

```
; merge: list-of-num [sorted] list-of-num [sorted]
; -> list-of-num [sorted]
(define (merge a b)
  (cond
    [(empty? a) b]
    [(empty? b) a]
    [else
     (cond
       [(< (first a) (first b))
        (cons (first a) (merge (rest a) b))]
       [else
        (cons (first b) (merge a (rest b)))])]))
```

Vector Mergesort

```
; mergesort: vector-of-num -> vector-of-num [sorted]
(define (mergesort vec)
  (define len (vector-length vec))
  (cond
    [(<= len 1) vec]
    [else
     (define half (quotient len 2))
     (merge
      (mergesort (vector-copy vec 0 half))
      (mergesort (vector-copy vec half len)))]))
```

Vector Merge

```
; merge: vector-of-num [sorted] vector-of-num [sorted]
; -> vector-of-num [sorted]
(define (merge a b)
  (define alen (vector-length a))
  (define blen (vector-length b))
  (define dlen (+ alen blen))
  (define dest (make-vector dlen))
  (for/fold ([ai 0] [bi 0]) ([di (in-range dlen)])
    (cond
      [(or (= bi blen)
           (and (< ai alen)
                (< (vector-ref a ai)
                   (vector-ref b bi))))
       (vector-set! dest di (vector-ref a ai))
       (values (add1 ai) bi)]
      [else
       (vector-set! dest di (vector-ref b bi))
       (values ai (add1 bi))]))
  dest)
```

Quicksort

Quicksort is an in-place sort for array:

- Pick a ***pivot*** value *n*

  conceptually, *↓ is always *n*

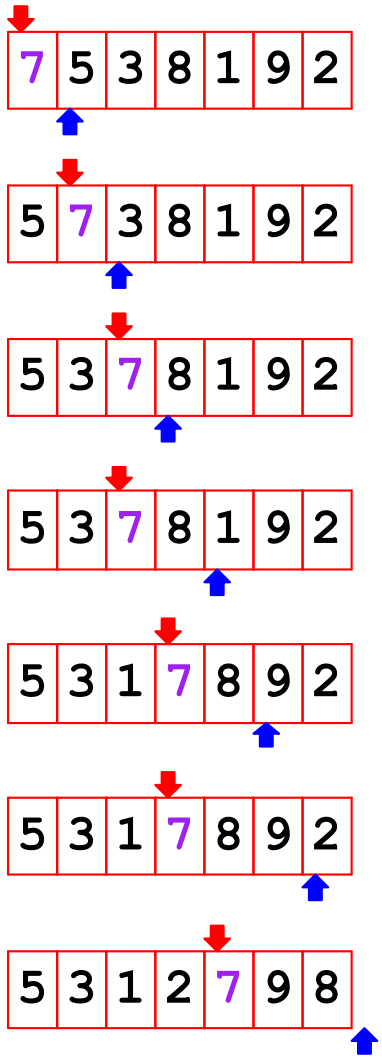
- For each element in the array, if it is less than the pivot, move it toward the front by *swapping*

  *↑ to *↓

  * (↓+1) to *↑

  conceptually, *n* to * (↓+1)

- The array is now two parts: all values less than the pivot and all values greater; sort the two parts (after which the whole array is sorted)



Quicksort

```
; quicksort! : vector-of-num num num -> void
; effect: sorts vec between n (incl) and m (excl)
(define (quicksort! vec n m)
  (when (> (- m n) 1)
    (define pivot (vector-ref vec n))
    (define pre
      (for/fold ([pre n]) ([i (in-range (add1 n) m)])
        (define v (vector-ref vec i))
        (cond
          [(< v pivot)
           (vector-set! vec pre v)
           (vector-set! vec i (vector-ref vec (add1 pre)))
           (values (add1 pre))]
          [else (values pre)])))
    (vector-set! vec pre pivot)
    (quicksort! vec n pre)
    (quicksort! vec (add1 pre) m)))
```